Outline

- heap sort
- priority queues
- hash tables
- hash functions
- open addressing
- chained hashing

heap sort

- start at the front of an unsorted array, and build a max heap with each element in turn
  - the heap expands exactly as the unsorted part of the array shrinks
- once the max heap is built, repeatedly delete the largest element from the heap and put it in its sorted place
  - the heap shrinks exactly as the sorted part of the array grows.

partial example of heap sort: building the heap

11	18	16	12	15	17	subarray of size 1 is always a heap
18	11	16	12	15	17	swap new value with parent -- done!
18	11	16	12	15	17	new value is less than parent -- done!
18	12	16	11	15	17	swap new value with parent -- done!

partial example of heap sort: removing from the heap

18	17	16	12	11	15	
15	17	16	12	11	18	swap largest with last value
17	15	16	12	11	18	swap root with largest child -- done!
11	15	16	12	17	18	swap largest with last value
16	15	11	12	17	18	swap root with largest child -- done!

priority queues

- the queues studied so far were strictly FIFO
- that means objects were returned in the order inserted
- in the real world, queues may need priorities, e.g.:
  - at airport check-in, there is a special line for first-class passengers
  - if any first-class passengers are waiting, they are handled first
  - if no first-class passengers are waiting, only then will the check-in agent check in the other passengers
- likewise some kinds of traffic in a network get priority
  - e.g. identified real-time traffic on WiFi

priority queue implementations

- linked list: objects are inserted in the proper place in the list in O(n), objects are returned from the front of the list in O(1)
  - nice because insertion of high priority items is fast
- array: objects are inserted in the proper place in the array, with all other objects shifted to make room in O(n), objects are returned from the back of the array in O(1)
- binary search tree: objects are inserted into the binary search tree, using the priority as the key in O(n), objects are removed from the leftmost or rightmost node of the tree in O(n)
  - with a balanced tree algorithm is used, the times become O(log n)
- heap: objects are inserted into the heap using the priority as the key in O(log n), and removed from the top of the heap in O(log n)
  - this is the simplest algorithm with guaranteed log time operations

priority queue performance

- if the priority queue only has a few elements, any of these implementations is fine
- however, if the priority queue might grow long, then frequent operations should be fast
- the performance depends on the algorithm, on the operation, and on the priority
- e.g., always adding something with highest priority is fast if using a linked list

priority queue in a heap

- O(log n) insertion and removal
- the priority is used as the key that determines the sorting order
- the value to be returned also has to be stored in the heap
- values with the same priority are returned in random order
  - not necessarily in FIFO order
  - unless we keep a count of insertions, and use it as part of the priority
- example: adding to a priority queue implemented as a heap the values a, b, c with priority 100 and x, y, z with priority 5
- first 3 removals give cab or acb or bca or abc or cba or bac

hashing

- hash browns: mixed-up bits of potatoes, cooked together
- hashing: slicing up and mixing together
- a hash function takes a larger, perhaps variable amount of data and turns it into a fixed-size integer
  - simple example: add all the bytes in a byte array modulo 256
  - the final value is between 0 and 255
- in-class exercise: why is this useful?

hash tables

- goal: given keys and values, store the values in an array in a location specified by the key
    - lookup is O(1), add/insert/set is O(1)
- but we can only use an array if the key is a small integer, <= the size of the array
- hash functions take arbitrary keys and turn them into small integers
- so we can:
    - use the hash of the key to
    - index the array, to
    - find out if the element is present, and if so,
    - get its value
- all this in constant time!
- this is a hash table

hash table

- a hash table is a collection class that under optimal conditions (best case) gives constant access time to elements in the collection
- a perfect hash function maps each key to a different array location
  - for example, if I am writing a compiler for a specific language, the language will have a fixed set of keywords, and I may be able to find a perfect hash function that maps each of these keywords to a different location in a small array
  - the perfection of this hash function depends on the array size
  - with a perfect hash function, looking up values in a hash table and adding values to a hash table are both O(1)
- in real life, perfect hash functions are hard to find, in part because the keys may not be known in advance
- if different keys map to the same location, that is a collision 🚗💥🚙

hash table example

- use the sum of the characters in a string as its hash
- use 1 for "a", 2 for "b", etc
- so the string "edo" hashes to 5 + 4 + 15 = 24
- the string "hello" hashes to 8 + 5 + 12 + 12 + 15 = 52
- the string "world" hashes to 23 + 15 + 18 + 12 + 4 = 72
- with a hash table of size 11, this is a perfect hash function for these strings:
  - 24 % 11 = 2, so "edo" is stored at index 2
  - 52 % 11 = 8, so "hello" is stored at index 8
  - 72 % 11 = 6, so "world" is stored at index 6

hash table example, continued

- supposing I wanted to use the same hash function on a table of size 3,
  - 24 modulo 3 = 0, so "edo" is stored at index 0
  - 52 modulo 3 = 1, so "hello" is stored at index 1
  - 72 modulo 3 = 0, so "world" is stored at index 0 🚗💥🚙
- the first and last string now need to be stored in the same location, which is a collision
  - an array location can store at most one element!!!!

hash functions

- finding a perfect hash function is only possible if all keys are known in advance
- for random, evenly distributed keys, good hash functions produce random, evenly distributed hash codes, with few collisions
- for non-random keys that resemble one another (e.g. edo1, edo2, edo3), good hash functions still produce random, evenly distributed hash codes
- as an example of a <u>bad</u> hash function, using the first character of a string gives more collisions than a more random hash function, unless all characters are equally likely
- in-class exercise: using h(key) = key mod table size, insert elements with key 99, 43, 14, 77 into a table of size 10

practical hash functions

- for a much more in-depth explanation of hash functions, see http://burtleburtle.net/bob/hash/doobs.html

  which includes a link to a more effective (and more complex) hash function:

  http://burtleburtle.net/bob/c/lookup3.c

- these hash functions use operations such as

  1. a << n, which shifts the integer a to the left by n bits
     - equivalent to $a*2^n$, and
  2. a ^ b, which computes the bit-wise xor of a and b
     - xor doesn't lose any information: each bit of the input affects the output
     - in contrast, if a is 0, a && b loses information about the value of b

hash functions in Java

- the Java Object class has a hashCode method:

```
int hashCode()
```

- this means every object in Java has a built-in hash function
- the hashCode method built-in to the Object class returns an int derived from the address of the object
- hashCode should return the same for any two object for which .equals returns true
  - so any object that implements its own .equals should also implement .hashCode
- for example, the String class computes a hash code from the characters and their position in the string

cryptographic hash functions

- for a hash table, the most important property of a hash function is that each key be mapped (as much as possible) to a different index
- knowing the hash function, a clever person can create a key that maps to a given index
  - for example, if I want a string that hashes to index 10, I can create the key "df" whose hash is 4+6=10
- with some hash functions, this is hard
  - e.g. if changing any bit of the input changes about ½ the bits of the hash
- such cryptographic hash functions are useful for identifying data

cryptographic hash functions in practice

- checking the hash of an input (for example, a file) can give assurance that the input has not been modified
- SHA-3 and SHA-2 cryptographic hashes give 224-bit, 256-bit, 384-bit, or 512-bit hash values
  - bitcoin miners repeatedly compute the SHA256 of a block until the hash is less than a given number
- earlier cryptographic hash functions include SHA-1, SHA-0, MD5 and more
  - for these (now largely obsolete) hash functions, it may be possible for an attacker to create a document that has a specific hash value

hash table collisions

- there are two main sources of collision in a hash table:
  - hash function collision: two different keys hash to the same integer
    - if adding all the letters, “edo” and “doe” both hash to 24, which is an inevitable collision
      - the example function is not a very good hash function!
      - a cryptographic hash function makes it very hard to find data that collides
  - hash table array index collision: two different integers, modulo the hash table length, give the same index
    - when adding all the letters of the keys “edo”, “hello”, and “world”, table size 11 has no collisions, table size 3 has collisions
    - so resizing the hash table may change the number of hash table collisions (but not predictably)

hash table load factor and collisions

- the load factor of a hash table is the number of elements divided by the table size
- as the load factor approaches 100%, collisions are more likely

ways to handle hash table collisions

- an array location can only hold one item, so what to do in case of a collision?
- three main solutions:
  - can increase (e.g. double plus one) the size of the array until all collisions go away
    - only works if the hash function itself has no collisions, and may use a lot of memory
  - can have each array element refer to a linked list rather than a single element: chained hashing
  - can look for another, unused place in the same array: open address hashing
- in-class exercise: assuming few collisions, what is the average runtime of find and add for each of the above strategies?
- in-class exercise: assuming many collisions, what is the worst-case runtime of find and add for each of the above strategies?

hash table runtime analysis

- three main solutions:
  - increase the size of the array until collisions go away
  - chained hashing: each array element refer to a linked list of values
  - open address hashing: look for an array location with no element
- in-class exercise: assuming few collisions, what is the average runtime of find and add for each of the above strategies?
- in-class exercise: assuming many collisions, what is the worst-case runtime of find and add for each of the above strategies?

chained hashing

- chaining or chained hashing: each array element refers to a linked list of elements

```
LinkedNode<T> hashTable[]
```

- with a really bad hash function (e.g. a function that always returns the same integer) all the data is stored in one linked list
- with good hash functions, most lists will be short
- the load factor can exceed 100%
- requires dynamic memory allocation
  - fine for Java, less interesting for languages such as C or inside the operating system
- over half the storage is for links rather than data, so inefficient for very large hash tables
- when there is no concern about space usage and about dynamic memory allocation, chained hashing is the preferred method for resolving collisions

open addressing

- when inserting a value in a hash table, if the slot (array location) indicated by the hash function is full, insert it into another slot
- this works until the hash table is full (100% load factor), i.e. it works as long as there are open slots
- the probe sequence determines where to look next when there is a collision
- when looking up a value in a hash table, the same probe sequence must be followed as when inserting

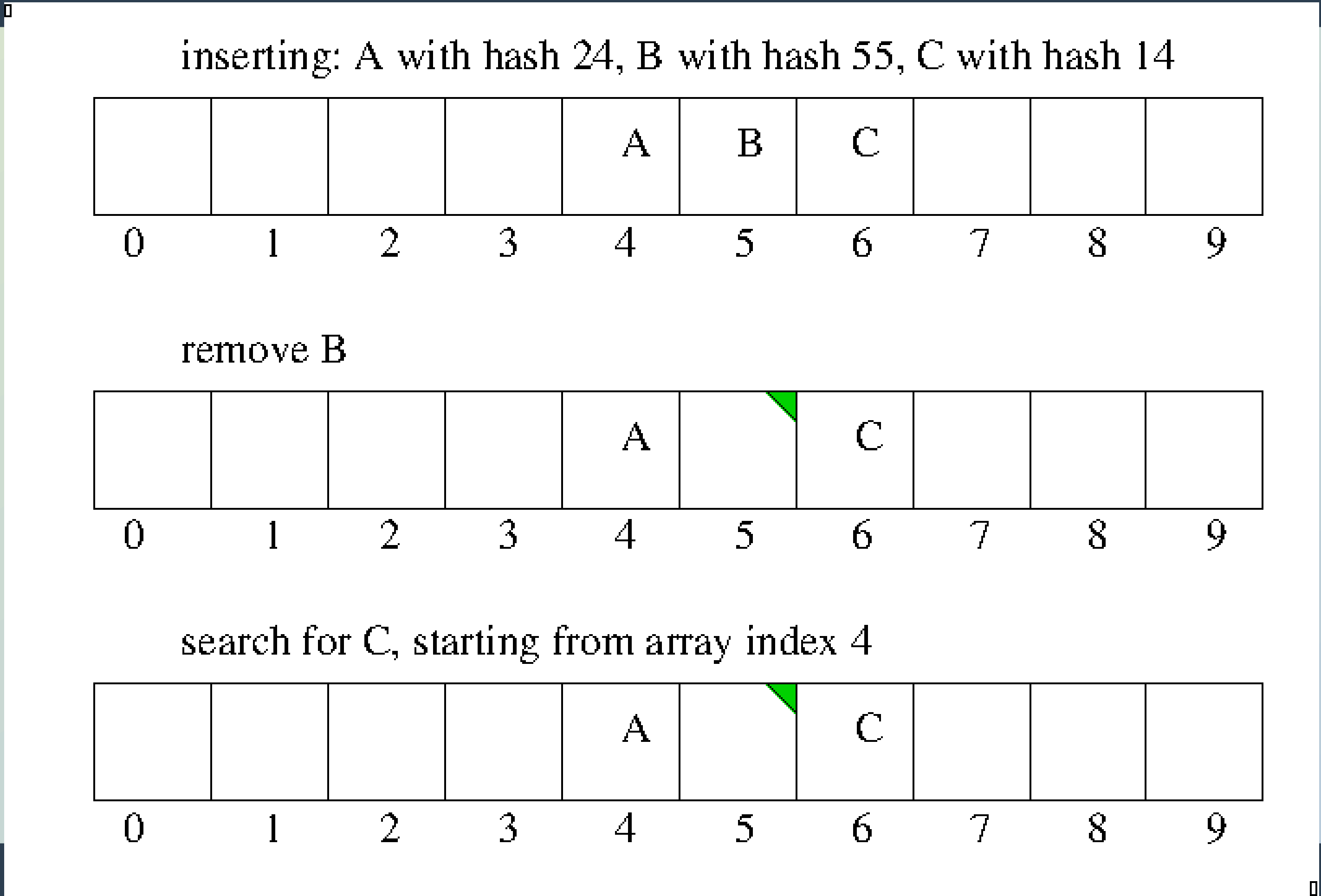
probing in open addressing

- assume a collision at index i
- linear probing: look at subsequent locations for an open slot: i, i + 1, i + 2, i + 3, i + 4, ...
- double hashing: a second hash function determines the step size: if the second hash function gives me h, look in locations i, i + h, i + 2h, i + 3h, i + 4h, … (mod the table size)
- double hashing avoids the bunching up of data at popular indices of the hash table
  - as long as any collisions are on the table index rather than the hash function

open addressing with deletion

- some hash tables only allow adding values, while others also allow removing values
- when removing a value from a hash table with open addressing, the hash table must record that the element was removed, so future searches can keep looking when they reach the slot of a deleted element
- each slot must record whether it is empty, full, or deleted
- a slot can only be empty until the first time a value is inserted, after which it can only be full or deleted

open addressing with deletion: example



linear probing (linear hashing)

- increment the index (modulo the array size) until a free slot is found
- with unevenly-distributed keys and hash functions that mostly give numbers close to each other, all the values are stored next to each other in the array
- this may lead to long search and insertion times

open addressing table size

- load factor cannot exceed 100%, so the table must have at least as many slots as the number of stored elements
- if the table size is a prime number, linear hashing or double hashing will visit the entire table before giving up
- otherwise, for example double hashing in a table of size 100, with step size 10, can only visit 1/10th of the slots
- if the table size is ever changed, each element must be reinserted using the recomputed hash function and the new table size
  - since just copying the old array would map an element to the wrong index:

    24 modulo 11 = 2, but 24 modulo 23 = 1
- for the same amount of data, the overall storage used is less than chained hashing